Simulating the world with Abstraction

C++ bindings for OpenCL

Jayanth Gummaraju
Benedict R. Gaster

SIGGRAPHASIA2009
the pulse of innovation

Motivation

“In my experience, C++ is alive and well -- thriving, even. This surprises many people. It is not uncommon for me to be asked, essentially, why somebody would choose to program in C++ instead of in a simpler language with more extensive "standard" library support, e.g., Java or C#.”

Scott Meyer

- Data abstraction
- Object-oriented programming
- Generic templates

Goals

- Lightweight, providing access to the low-level features of the original OpenCL C API.
- Compatible with standard C++ compilers (GCC 4.x and VS 2008).
- Not all C++ features are supported by default
- Should not require the use of the Standard Template Library.
- The bindings should be defined completely within a header, cl.hpp.

Something simple – hello from OpenCL C++

```
#define __CL_ENABLE_EXCEPTIONS
#define __NO_STD_VECTOR
#define __NO_STD_STRING
#if defined(__APPLE__) || defined(__MACOSX)
#include <OpenCL/cl.hpp>
#else
#include <CL/cl.hpp>
#endif
#include <cstdio>
#include <cstdlib>
#include <iostream>

const char * helloStr = "__kernel void  hello(void) { }\n";
```

Something simple – “hello” from OpenCL

```
int main(void) {
    try {
      cl::Context context(CL_DEVICE_TYPE_GPU, 0, NULL, NULL, &err);
      cl::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();
      cl::CommandQueue queue(context, devices[0], 0, &err);
      cl::Program::Sources source(1, std::make_pair(helloStr,strlen(helloStr)));
      cl::Program program_ = cl::Program(context, source);
      program_.build(devices);
      cl::Kernel kernel(program_, "hello", &err);
      cl::KernelFunctor func = kernel.bind(queue, cl::NDRange(4, 4), cl::NDRange(2, 2));
      func().wait();
    } catch (cl::Error err) {
      std::cerr << "ERROR: " << err.what() << "(" << err.err() << ")“  << std::endl;
    }
    return EXIT_SUCCESS;
}
```

Real Time Ocean Simulation

- FFTs are a common example
  – Computationally expensive
  – Data parallel

- Today's GPUs are extremely good at executing FFTs:
  – 1k x 1k is easy ☺
  – 2k x 2k is easy but real time rendering becomes interesting (~2Million polygons)

Real Time Ocean Simulation

- Jerry Tenssendorf's work:
  - Simulating Ocean Water, SIGGRAPH 1999
  - Waterworld, Titanic, and many others (2kx2k FFTs)
  - Works with sums of sinusoids but starts in Fourier domain
  - Can evaluate at any time t without having to evaluate other times
  - Use the Phillips Spectrum
    - Roughness of waves is a function of wind velocity

- Jason L. Mitchell's work:
  - Real-Time Synthesis and Rendering of Ocean Water, 2005
  - DX-9 Demo 256x256 FFTs, low and high frequencies separated

Real Time Ocean Simulation

Real Time Ocean Simulation

- Starts with a platform…

```
cl::vector<cl::Platform> platforms;
err = cl::Platform::get(&platforms);

checkErr(err && (platforms.size() == 0 ? -1 : CL_SUCCESS), "cl::Platform::get()");

// As cl::vector (implements std::vector interface) straightforward to determine number of
// platforms, no need for additional variables as required by clGetPlatformIDs.
std::cout << "Number of platforms:\t " << platforms.size() << std::endl;

for (cl::vector<cl::Platform>::iterator i = platforms.begin(); i != platforms.end(); ++i) {
    // pick a platform and do something
    std::cout << "  Platform Name: " << (*i).getInfo<CL_PLATFORM_NAME>().c_str()
            << std::endl;
}
```

Real Time Ocean Simulation

- clGetXInfo functions are provided in two flavors
  - A static version of the form:

```
template <cl_int name> typename   detail::param_traits<detail::cl_device_info,
    name>::param_type    getInfo(cl_int* err = NULL) const
```

  - A dynamic version of the form:

```
template <typename T>   cl_int getInfo(cl_device_info name, T* param) const
```

- Unlike the C API the C++ bindings return info values directly:
  - No need to call info function to find memory requirements
  - Mapping from **cl_X_info** enums to C++ types, so for **cl_platform_info**:

```
F(cl_platform_info, CL_PLATFORM_PROFILE, STRING_CLASS) \
F(cl_platform_info, CL_PLATFORM_VERSION, STRING_CLASS) \
F(cl_platform_info, CL_PLATFORM_NAME, STRING_CLASS) \
F(cl_platform_info, CL_PLATFORM_VENDOR, STRING_CLASS) \
F(cl_platform_info, CL_PLATFORM_EXTENSIONS, STRING_CLASS)
```

Real Time Ocean Simulation

- Continue with a context of devices…

```
  cl::vector<cl::Platform>::iterator p = platforms.begin();
  …
  // Get all devices supported by a particular platform
cl::vector<cl::Device> devices;
  (*p).getDevices(CL_DEVICE_TYPE_GPU | CL_DEVICE_TYPE_CPU, &devices);

// Create a single context for all devices
cl::Context context(devices, NULL, NULL, NULL, &err);
checkErr(err, "Conext::Context()");
  // Create work-queues for CPU and GPU devices
  queueCPU = cl::CommandQueue(context, devices[0], 0, &err);
checkErr(err, "CommandQueue::CommandQueue(CPU)");
  queueGPU = cl::CommandQueue(context, devices[1], 0, &err);
checkErr(err, "CommandQueue::CommandQueue(GPU)");
```

Real Time Ocean Simulation

- Load kernel sources and build programs…

```
std::ifstream file(“ocean_kernels.cl");
checkErr(file.is_open() ? CL_SUCCESS : -1, "reading ocean_kernels.cl");
std::string prog(std::istreambuf_iterator<char>(file), (std::istreambuf_iterator<char>()));

cl::Program::Sources source(1, std::make_pair(prog.c_str(),prog.length()+1));
                    cl::Program program(context, source);
err = program.build(devices);

if (err != CL_SUCCESS) {
      std::cout << "Info: " << program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(devices);
      checkErr(err, "Program::build()");
}
```

Real Time Ocean Simulation

- One kernel for generating Phillips Spectrum:

```
__kernel  void phillips(__global float2 * buffer, __global float2 *const randomNums,
                        float windSpeed, float windDir,  unsigned int height, unsigned int width)
```

- Two (data) parallel kernels for FFT:

```
__kernel __attribute__((reqd_work_group_size (64,1,1)))
void kfft(__global float *greal, __global float *gimag)

__kernel __attribute__((reqd_work_group_size (64,1,1)))
void  ktran(__global float *greal, __global float *gimag)
```

- One kernel for calculating the partial differences, i.e slopes, that is used to calculate the normal's from the height map for light shading:

```
__kernel void calculateSlopeKernel(__global float* h, __global float2 *slopeOut, uint width, uint height)
```

Real Time Ocean Simulation

- Create kernel objects…

```
phillipsKernel = cl::Kernel(program, ”kphillips", &err);
checkErr(err, "Kernel::Kernel(kphillips)");

kfftKernel = cl::Kernel(program, "kfft", &err);
checkErr(err, "Kernel::Kernel(kfft)");

ktranKernel = cl::Kernel(program, "ktran", &err);
checkErr(err, "Kernel::Kernel(ktrans)");

calculateSlopeKernel = cl::Kernel(program, “calculateSlopeKernel", &err);
checkErr(err, "Kernel::Kernel(calculateSlopeKernel)");
```

Real Time Ocean Simulation

- Allocate memory buffers…

```
imag = cl::Buffer(context, CL_MEM_READ_WRITE,1024*1024*sizeof(float), 0, &err);
checkErr(err, "Buffer::Buffer(imag)");

spectrum = cl::Buffer(context, CL_MEM_READ_WRITE, 1024*1024*sizeof(cl_float2), 0, &err);
checkErr(err, "Buffer::Buffer(spectrum)");

// Height map and partial differences (i.e. slopes) generated directly into GL for renderning
real = cl::BufferGL(context,CL_MEM_READ_WRITE, heightVBO,&err);
checkErr(err, "BufferGL::BufferGL(height)");

slopes = cl::BufferGL(context, CL_MEM_READ_WRITE, partialDiffsVBO, &err);
checkErr(err, "BufferGL::BufferGL(partialDiffs)");
```

Real Time Ocean Simulation

- Do the work…

```
phillipsEvent.wait(); // make sure spectrum is up-to-date, i.e. account for wind changes. in practice
                        // we double buffer to avoid causing delays due to continual wind changes
cl::vector<cl::Memory> v;
v.push_back(real); v.push_back(partialDiffs);
err = queueGPU.enqueueAcquireGLObjects(&v);
checkErr(err, "Queue::enqueueAcquireGLObjects()");
err = kfftKernel.setArg(0, real); // other arguments are invariant and set once during setup
err = queueGPU.enqueueNDRangeKernel(kfftKernel, cl::NullRange,
                              cl::NDRange(1024*64), cl::NDRange(64));
checkErr(err, "CommandQueue::enqueueNDRangeKernel(kfftKernel1)");
err = ktranKernel.setArg(0, real);  // other arguments are invariant and set once during setup
err = queueGPU.enqueueNDRangeKernel(ktranKernel, cl::NullRange,
                              cl::NDRange(128*129/2 * 64), cl::NDRange(64));
checkErr(err, "CommandQueue::enqueueNDRangeKernel(ktranKernel1)");
```

Real Time Ocean Simulation

- Do the work (cond)…

```
// note, no need to set argument as they persist from previous calls
err = queueGPU.enqueueNDRangeKernel(kfftKernel, cl::NullRange,
                                    cl::NDRange(1024*64), cl::NDRange(64));
checkErr(err, "CommandQueue::enqueueNDRangeKernel(kfftKernel2)");

err  = calculateSlopeKernel.setArg(0, real); err |= calculateSlopeKernel.setArg(1, slopes);
err |= calculateSlopeKernel.setArg(2, width); err |= calculateSlopeKernel.setArg(3, height);
checkErr(err, "Kernel::setArg(calculateSlopeKernel)");

err = queueGPU.enqueueNDRangeKernel(calculateSlopeKernel, cl::NullRange,
                                    cl::NDRange(width,height), cl::NDRange(8,8));
checkErr(err, "CommandQueue::enqueueNDRangeKernel(calculateSlopeKernel)");
```

Real Time Ocean Simulation

- Do the work (cond)…

```
err = queueGPU.enqueueReleaseGLObjects(&v);
checkErr(err, "Queue::enqueueReleaseGLObjects(GPU)");
queueGPU.finish();
```

- And when the wind changes…

```
queueCPU.enqueueTask(phillipsKernel, NULL, &phillipsEvent);
```

Optional features - Exceptions

- Not enabled by default
- To enable define the following before including cl.hpp:
  - #define __CL_ENABLE_EXCEPTIONS
- API calls that originally return an "*cl_int err*" will now throw the exception, *cl::Error*,on error:

```
catch (cl::Error err)
{
    std::cerr << "ERROR: " << err.what()
              << "(" << err.err() << ")" << std::endl;
}
```

- err.what() returns an error string.
- err.err() returns the error code.

Optional features – No STL usage

- Not everyone wants to use STL in their applications.
- OpenCL C++ bindings use std::string and std::vector.
- Provide alternative implementations:
  - cl::string and cl::vector
- Allow user defined versions.

Try them out…

- Planned for adoption as part of OpenCL 1.1
- Can be downloaded from Khronos OpenCL site:
  - http://www.khronos.org/registry/cl/
- Multiple platform and cross vendor:
  - A little abstraction goes a long way!